1. Circuit Satisfiability Problem

(1) What have you done?

① 利用 MPI 將工作平分給不同的 process

② 利用 MPI_Recv 和 MPI_Send 將每個 process 計算完的結果以 tree structured 的型式互相傳遞。

③ 最後依 rank = 0 的 process 統整結果，並計算時間及答案。

(2) Analysis on your result

① 未平行化，平行化比較：

未平行化：

```
for (i = 0; i <= USHRT_MAX; i++) {
   count += checkCircuit (id, i);
}
```

就是一個正常的迴圈，沒有其它的process參與，單一個process從頭做到尾。

平行化：

```
// Determine the number of processors
int number_of_processes;
MPI_err = MPI_Comm_size(MPI_COMM_WORLD, &number_of_processes);

// Determine the rank of this processor
int MPI_rank; /* process id */
MPI_err = MPI_Comm_rank(MPI_COMM_WORLD, &MPI_rank);
```

利用 MPI_Comm_size 及 MPI_Comm_rank 分別計算出process的數量和每個process自己的 rank。

```
for (i = MPI_rank; i <= USHRT_MAX; i += number_of_processes) {
        MPI_sum += checkCircuit(MPI_rank, i);
}
```

帶不同的process進來，就會去執行他們自己的工作，以process數為8舉例，rank為1的process就會執行 i = 1, 9, 17, 25, 32... 直到USHRT_MAX為止的工作。

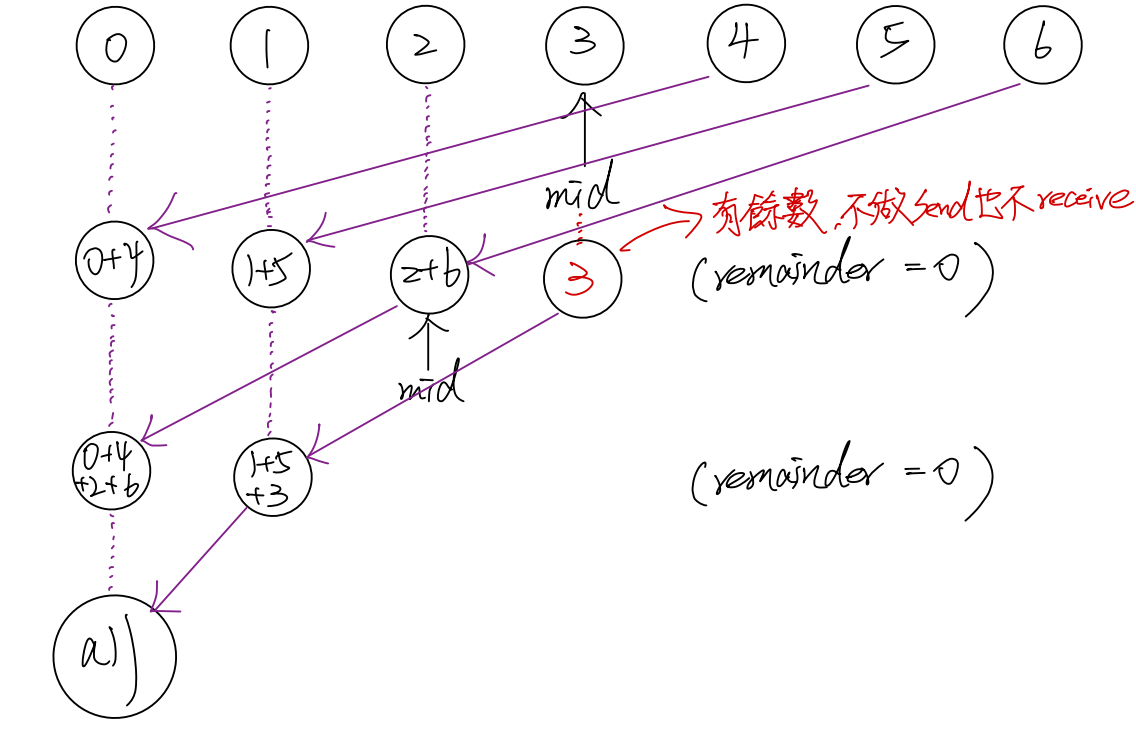
② tree structured：

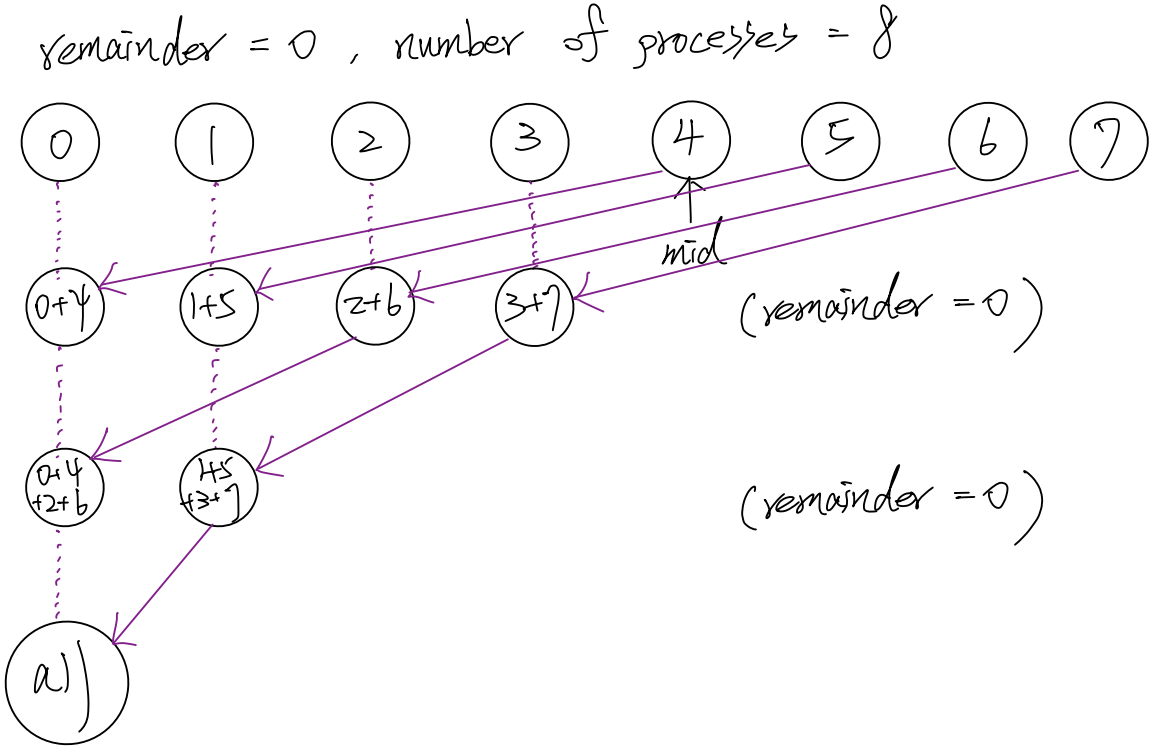
```c
void MPI_calculate(int MPI_rank, int number_of_processes, int *MPI_sum, int *MPI_local_count) {
    while(number_of_processes != 1) {
        int mid = number_of_processes/2;
        int remainder = number_of_processes%2;
        if(MPI_rank < mid) {
            MPI_Recv(MPI_local_count, 1, MPI_INT, MPI_rank+mid+remainder, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                *MPI_sum += *MPI_local_count;
                }
        else if(MPI_rank >= mid+remainder && MPI_rank < number_of_processes) {
            MPI_Send(MPI_sum, 1, MPI_INT, MPI_rank-mid-remainder, 0, MPI_COMM_WORLD);
            return;
        }
        number_of_processes = mid+remainder;
    }
}
```

考慮到process數量可能不是2的次方，就加上3餘數的部分。

以process數量為7和8舉例：

remainder = 1, number of processes = 7





③ 執行結果：

process數	執行時間(s)	執行結果(number of solutio...
1	0.001779	9
2	0.000851	9
3	0.000835	9
4	0.000492	9
8	0.000361	9
9	0.015638	9
16	0.051719	9
27	0.068081	9
32	0.103749	9
40	0.152033	9

當 process 數量為 1 時，即為未平行化的 serial program。而當 process 數往上增加時，執行時間

也隨之減少，可以看出平行化的成效。

當process數來到9時，時間就比serial還要慢，這裡推估9可能是一個threshold，或著也有可能數量不是2的次方造成效能減低。

而當process再增加時，時間就只增不減，這裡應該就是process之間的溝通時間超過正常的執行導致的overhead。

(3) Difficulties

整個MPI的使用在一開始完全不知道如何下手，而之後還要配合tree structured做接收傳遞，非常困難，不過知道了運作流程就可以寫出來。

2. Monte Carlo Method

(1) What have you done?

① 利用 MPI 將工作平分給不同的 process

② 利用 MPI_Recv 和 MPI_Send 將每個 process 計算完的結果以 tree structured 的型式互相傳遞。

③ 最後依 rank = 0 的 process 統整結果，並計算時間及答案。

(2) Analysis on your result

① 未平行化，平行化比較：

未平行化：

```
number_in_circle = 0
for (toss = 0; toss < number_of_tosses; toss++){
    x = random double between -1 and 1;
    y = random double between -1 and 1;
    distance_squared = x*x + y*y;
    if (distance_squared <= 1) number_in_circle++;
}
pi_estimate = 4 * number_in_circle/((double)number_of_tosses)
```

平行化：

```
// Determine the number of processors
int number_of_processes;
MPI_err = MPI_Comm_size(MPI_COMM_WORLD, &number_of_processes);

// Determine the rank of this processor
int MPI_rank; /* process id */
MPI_err = MPI_Comm_rank(MPI_COMM_WORLD, &MPI_rank);
```

利用 MPI_Comm_size 及 MPI_Comm_rank 分別計算出 process 的數量和每個 process 自己的 rank。

```
for (i = MPI_rank; i <= number_of_tosses; i += number_of_processes) {
        x = (double)rand() / RAND_MAX * 2 - 1;
        y = (double)rand() / RAND_MAX * 2 - 1;
        distance_squared = POS(x, y);
        if (distance_squared <= 1) {
                number_in_circle += 1;
        }
}
```

而不同的 process 進來，就會去執行他們自己的工作，這邊的工作即為計算在圓裡面點的數量。

② tree structured：

```
void MPI_calculate(int MPI_rank, int number_of_processes, ll *number_in_circle) {
    while(number_of_processes != 1) {
        ll MPI_local_count;
        int mid = number_of_processes/2;
        int remainder = number_of_processes%2;
        if(MPI_rank < mid) {
            MPI_Recv(&MPI_local_count, 1, MPI_LONG_LONG_INT, MPI_rank+mid+remainder, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            *number_in_circle += MPI_local_count;
        }
        else if(MPI_rank >= mid+remainder && MPI_rank < number_of_processes) {
            MPI_Send(number_in_circle, 1, MPI_LONG_LONG_INT, MPI_rank-mid-remainder, 0, MPI_COMM_WORLD);
            return;
        }
        number_of_processes = mid+remainder;
    }
}
```

這邊的 tree structured 和第一題為相同架構。

③ 執行結果：

先比較同樣N的數量，不同process數量的結果：

process數	執行時間(s)	執行結果(pi)
1	24.69882	3.141598932
2	11.398129	3.1415667
3	7.88584	3.14157625
4	6.614439	3.1414963
8	7.919856	3.141609248
9	7.451926	3.1417175
16	6.160031	3.141525636
27	6.088024	3.14114318
32	4.580331	3.14103782
40	3.952045	3.141660964

當process數量為1時，即為未平行化的serial program。而當process數往上增加時，執行時間也隨之減少，可以看出平行化的成效。

而process數在中間值時，大概都在6~8秒之間，直到process數到30~40，時間才再度減少。

再比較用樣process數為4時，不同N的結果

N	執行時間(s)	執行結果(pi)
1000	0.079861	3.684
10000	0.09201	3.1364
100000	0.043918	3.17444
1000000	0.068013	3.149764
10000000	0.044002	3.1459684
100000000	0.391994	3.1415328
1000000000	2.724016	3.141405924
10000000000	33.631965	3.141694494
1.0E+11	287.844062	3.141538715

可以看到N的數越大，在圓內的比例也會越接近面積比，π值也就越準確，不過伴隨而來的就是執行時間的大幅增加。

有些結果不符合預期，估計是C內建的random仍然沒有達到真正的隨機。

(3) Difficulties：

在做完第一題後，第二題基本上是參照著寫就行，因此沒有特別難的地方。

Feedback:

透過這次作業，可以了解MPI的實際操作，也能知道老師上課所說的內容。同時也理解這次作業助教非常仁慈，基本上只要寫好MPI和tree的部分就行，沒有額外的東西需要操作，謝謝助教！